

Project Report

Fast and accurate bug localization



August 25, 2016

Gabriel Pawlowsky

Mentor: Naveen Kulkarni, Sunil Vuppala

Co-Mentor: Nitin Agarwal

Infosys Automation Platform Unit

Contents

[Background 2](#_Toc459938774)

[Introduction 2](#_Toc459938775)

[Project Outline 3](#_Toc459938776)

[Project Report 4](#_Toc459938777)

[Phase 1 – Noun extraction 4](#_Toc459938778)

[Phase 2 – Keyword extraction 5](#_Toc459938779)

[Phase 3 – Concept identification 6](#_Toc459938780)

[Results 7](#_Toc459938781)

[Early filtration 7](#_Toc459938782)

[General purpose concept extraction 7](#_Toc459938783)

[Source code related concepts extraction 9](#_Toc459938784)

[Technically-focused Concepts extraction 9](#_Toc459938785)

[Usage 10](#_Toc459938786)

[Usage summary 11](#_Toc459938787)

# Background

## Introduction

Over the last 35 years, Infosys has maintained, operated and managed systems with global clients across every industry. Building on this deep experience, Infosys has recognized the need to bring artificial intelligence to the enterprise in a meaningful and purposeful way; in a way that leverages the power of automation for repetitive tasks and frees people to focus on the higher value work, and on breakthrough innovation. To realize this, organizations need to be able to do three things: manage organizational knowledge, apply it to automate enterprise processes, and utilize the massive intelligence hidden away in systems, machines and people. Today’s AI technologies address part of this with learning and information; with its Mana platform, Infosys is now bringing this together in a fundamental way with knowledge and understanding of the business and the IT landscape.

Infosys Mana is a knowledge-based AI platform. It brings machine learning together with the deep knowledge of an organization to drive automation and innovation. This enables businesses to continuously reinvent their system landscapes. Mana, with the Infosys AiKiDo service offerings, dramatically lowers the cost of maintenance for both physical and digital assets. It captures the knowledge and know-how of people across fragmented and complex systems, and simplifies the continuous renovation of core business processes. Mana also enables businesses to bring new, delightful user experiences leveraging state-of-the-art technology.

The Infosys Aikido framework that hosts the Mana platform helps companies undertake non-disruptive transformation of their existing landscapes:

* Ki: capturing the knowledge within legacy systems to renew, accelerate and enable them to bring entirely new experiences
* Ai: delivering open, intelligent platforms that bring transformation – new kinds of applications, software tools, unprecedented levels of data processing, a radical new cost performance
* Dō: design-led services which bring a Design Thinking approach that starts with a deep understanding of a client’s business and IT objectives, its users and customers, to find their most critical problems and biggest opportunities

Internally, Infosys Mana is comprised of three integrated components all of which are based on open source technology:

* Infosys Information Platform – an open source data analytics platform that enables businesses to operationalize their data assets and uncover new opportunities for rapid innovation and growth
* Infosys Automation Platform - a platform that continuously learns routing logic, resolution processes and diagnosis logic to build a knowledge base that grows and adapts to changes in the underlying systems
* Infosys Knowledge Platform – a platform to capture, formalize and process knowledge and its representation in a powerful ontology based structure that allows for the reuse of knowledge as underlying systems change

This project is designed to extend the Infosys Automation Platform (IAP) by additional functionality. Currently, in big companies developing large software products, errors in software execution are usually reported via three levels of tickets called L1, L2 and L3. As part of the IAP, Infosys is currently working on ways to automate the resolution process of such error reports on all levels to speed it up and require less man-power for it. Tickets resolved in the first two levels are usually related to problems occurring due to exceptions in the real-world execution of the software. Tickets reaching the third level, on the other hand, are usually related to bugs directly in the source code of the developed software which makes it extremely hard to fully automate their resolution without human intervention. Additionally, in such large corporations it is often not possible to forward the bug report to the actual developer of the software for various reasons including that the developer may have left the company or is needed in another project. Bringing in new people who are not initially familiar with the source code to resolve the issue, then, is often very time-consuming and may introduce additional costs such as unforeseen resulting bugs. This project is meant to be a first step towards reducing these costs by giving developers pointers to parts of the source code where a reported bug has most likely occurred to aid them in fixing it.

## Project Outline

Fast and accurate localization of software defects continues to be a difficult problem since defects can manifest as unexpected results at different locations. They are often intricate in nature and require persistent investigation. Layered architecture and multi-team or multi-module scenarios further compound the problem. Fixing issues is a time critical task. Hence, assisting developers with a set of possible buggy source code files can reduce the time taken by a developer to analyze a bug. Presently, developers often rely on their knowledge and simple search. However, such a knowledge is volatile and localized - limited to the files directly associated with a developer. With the knowledge of bug reports, this project aims at investigating how a staggered search with query reformulation can help identify buggy source files in a multi-team or multi-module scenarios. Essentially, the project consists of using various techniques from natural language processing to extract conceptual information from various sources such as the source code documentation, bug reports and resolutions of very similar bugs found online. The information from these different sources shall then be used to direct a new developer to those parts of the source code in which the bug is most likely to occur. Thereby, the developer should hopefully be able to fix the issue much more quickly.

The entire project is very large in scope and is taken on by a team of software developers. I had to implement one part of this large project. Mainly, this part is concerned with extracting useful concepts from the documentation of the source code in question that can then be integrated into the full program to build a computational understanding of the source code. Once bug reports and online resolutions have been analyzed in a similar way and a resolution for the actual bug at hand can be proposed, this concepts information will be used to direct the developer to the relevant parts of the source code.

The project is based on natural language processing and utilizes the various techniques publicly available to extract good quality concepts from programming source code documentations. In particular, the Programming and Streaming guides from the Spark documentation were used for testing purposes. The project was roughly split in three parts, namely noun extraction, keyword extraction and concept identification which will be further explained below.

# Project Report

## Phase 1 – Noun extraction

To get quality keywords or concepts using NLP techniques, one first needs the candidate terms which should be considered for their usefulness. This step is the noun extraction. Initially, I implemented this part in Java using the common NLP library OpenNLP. A working version of this implementation up to the lemmatization part (which will be explained shortly) can be found in the OpenNLPEngine.java file corresponding to this report. However, since OpenNLP does not support lemmatization, I switched to the Stanford CoreNLP library to use an all-in-one package. The latest version of my software, thus, uses the StanfordNLPEngine.java file instead of the aforementioned one containing the code for OpenNLP.

Having this distinction out of the way, let’s get into the details of the noun extraction phase. First, the documentations (for Spark, in our case) have to be read in programmatically, which is straightforward in the source code. Then, to get all the words of the text as individual tokens, I used the Stanford CoreNLP tokenizer split the text by spaces, dots and other commonly defined stop words. Now, to gain a better understanding of the text, I ran the Stanford CoreNLP POS (Part of Speech) – Tagger to assign a part of speech tag to every token. This is also a well-known NLP technique that can be simply accessed from the library which finds the right tags for each token by comparing them to an English language dictionary and considering grammar rules. This technique obviously works best when the sentences are well-formed in English but can be used in other cases with reasonable success as well. Finally, to further aid the following selection process, I used to Stanford CoreNLP Lemmatizer on all the tokens to put them into their canonical forms. That way, for instance, “go”, “went”, and “gone” would all be translated to “go” and, thus correspond to the same token. This may seem like a loss of information, but all these three words convey the same essential information for our concept extraction purposes and should increase the same text frequency count in our following selection process.

Using the POS-tags, I selected all unique nouns occurring in the text to be candidate terms, since all general concepts used to understand the source code will be nouns. Those nouns form the set of all u unigrams (single word tokens) considered to be candidate terms. Since a concept (such as “distributed dataset”) could also consist of multiple words, I chose to include bi- and tri-grams (two and three word tokens) featuring the following combinations of POS-tags which cover mostly all possibilities for concepts: for bigrams, I used verb-noun and noun-noun combinations, for trigrams I used verb-noun-noun and noun-noun-noun combinations of POS-tags for words occurring consecutively in the text.

After obtaining all these candidate terms, I did an early filtration to remove words which possess certain characteristics making them unlikely to be useful concepts. To do so, however, I first had to scan the text for each of the candidate terms to determine how often each keyword appears in the text after lemmatization. For unigrams, I then looked for the most frequently occurring token and discarded all tokens who occurred less than 5% or more than 70% as often as the maximum frequency word. This is a useful filtration since it removes words only occurring very infrequently (which suggests that they were just used in some explanation or side note or something of that sort and do not represent an important keyword in the text since those usually occur multiple times) and those occurring very frequently (such as “Spark” or “Java”, which are basic words that usually aren’t good concepts for our purposes). For the n-grams, instead, I used a t-statistics calculation to determine whether the words likely occurred in that order in the text because they belong together (like “distributed dataset”) or by chance (like “computer. Yesterday”). As suggested by my co-mentor Nitin Agarwal, I discarded n-grams with a t-value lower than 2.576.

At this point I had a clean list of candidate terms somewhat likely to be important concepts of the text and enough information about them to filter them further. The results up to this point can be found in the “Results”-section below.

## Phase 2 – Keyword extraction

A good keyword extraction scheme is usually based on computing scores of certain characteristics for each candidate term token where a higher score indicates a likely higher importance of this word in the text. The highest scoring words are then considered as keywords for the text. There are myriads of different approaches to calculate these scores. However, we generally decided on implementing an unsupervised approach (meaning that it does not require training for each new piece of text such as prepared texts with manually tagged keywords) to greatly reduce the time it takes for the system to work with a new type of source code documentation. This is very important to reduce the workload for the developers so they can really profit from this bug localization framework.

First I (almost-completely) implemented a Text Frequency-Inverse Document Frequency (TFIDF) scoring scheme. In TFIDF, the document is virtually split into different texts, for instance, by paragraphs. The TFIDF score is then calculated as the product of the frequency of a token occurring in this particular paragraph and the inverse of the frequency of the token occurring in the entire document. This score is, thus, highest for words that occur frequently in one paragraph but not so many times throughout the entire document. A candidate term with high TFIDF score is likely to be a keyword since keywords are often talked about in one paragraph where they are explained but not really throughout the entire document. However, I found that there are many libraries out there implementing different version of TFIDF and I can test more different approaches quickly by using these libraries as opposed to coding every version myself. Therefore, I switched to public libraries for the TFIDF calculation for later parts of my analysis.

Then, since scoring is probably by far the hardest part, I did a great deal of research on available scoring techniques and libraries doing the same. I evaluated many techniques such as BM25, location-in-document, Approximate Dictionary-Based Chunking, graph-based approaches, etc. In addition to that, I tested our documentations on many, many fully-fledged general purpose keyword extraction libraries including TextRank, JATE2.0, RAKE, LingPipe, AlchemyAPI (great results but could not be used due to it being commercial). Out of these, by manual examination of the extraction results for the aforementioned Spark documentations, JATE2.0 and RAKE turned out to give the best output for our purpose.

Out of those two, I decided to focus most of my time on working with JATE2.0, since it offers many different scoring techniques to choose from and great customizability of those techniques. A schematic overview for how JATE2.0 works is depicted in Figure 1. In particular, JATE2.0 uses the Apache Solr indexing framework to extract candidate terms for the text it is fed. This extraction together with virtually everything else JATE2.0 does can be calibrated in great detail via huge XML-configurations. In particular, one can choose from three techniques for the candidate term extraction: Part-of-Speech (PoS) pattern based, Noun Phrase (NP) chunking based, N-gram based. Of these, the POS-pattern based approach (which extracts the candidates by looking for patterns such as “This is a XXX” to signify keywords) gave the best results for our purpose because the other two rely on dictionaries and programming documentation often feature words like “Spark” that are hardly found in dictionaries but may well be useful concepts. By the way, the way I set up the JATE2.0 configuration, Apache Solr will also strip HTML-files of all their HTML characters and remove all the tags so that they can be automatically analyzed as well. Otherwise, text or even Microsoft Word and PDF files are supported. Furthermore, JATE2.0 offers the following techniques for scoring those candidate terms to identify keywords: TTF, ATTF, TTF-IDF, RIDF, C-value, ChiSquare, RAKE, Weirdness, GlossEx, TermEx.

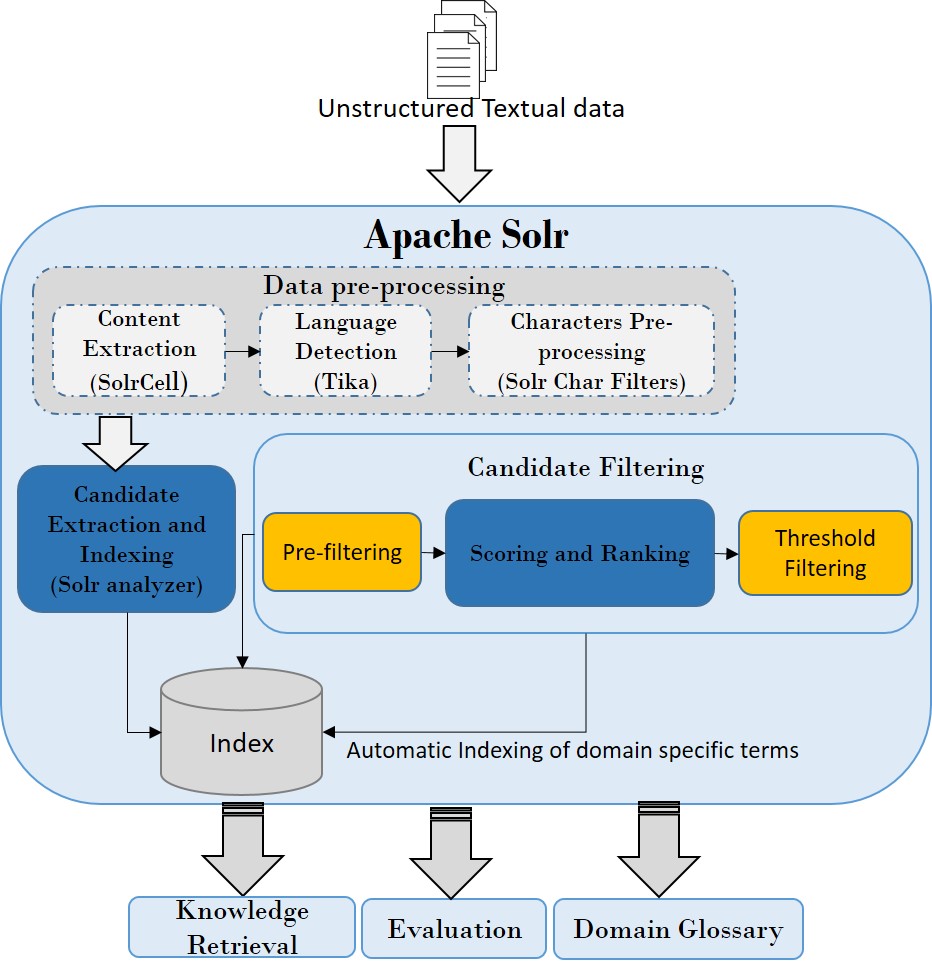


Figure 1: Schematic overview of JATE2.0 built on Apache Solr

Of these, TTF, ATTF, C-value and ChiSquare returned very simple, general keywords and mostly unigrams. Specifically, ATTF seemed to give the best overall results and C-value also gave some reasonably good results including some bigrams not present with ATTF. TTF-IDF and RIDF are both based on the inverse document frequency as outlined above and give, therefore, more technical parts specific to certain parts of the documentation. Weirdness, GlossEx and TermEx all rely strongly on dictionaries of good words and use different approaches to filter words from the text that are very different or “weird” in comparison to the dictionary terms. As expected, there techniques return very technical, usually source code related, concepts. Finally, RAKE uses a very popular but rather complex algorithm for automatic keyword extraction and gave very good results for technical concepts are well. However, the JATE2.0 implementation of the RAKE algorithm seemed to do worse than the original RAKE implementation in python so I used to original implementation for this technique.

## Phase 3 – Concept identification

This last phase is all about looking at the scores for candidate terms previously computed and using them to choose the best set of useful concepts possible. Useful concepts in this case mean that they should be useful for our purposes of understanding the nature of the software underlying the analyzed documentation. In this sense, for instance, “task” could be a useful concept if there is something in the software that corresponds to this task such as a class but it would not be useful if it is just used to convey another concepts such as in a construct like “The task of this controller is…”. Also, generally, “Java” could be a keyword of the text but it would not help us understand the particular source code we are dealing with and therefore would not really be a useful concept. Consequently, the best algorithm to choose useful concepts strongly relies on the particular use-case at hand. For this reason, I have developed three different combinations of scores to consider to identify the concepts from the scored candidates. The way in which these three schemes work is outlined together with the results in the following section.

# Results

## Early filtration

As mentioned before, I would first like to present the results of the pure noun extraction after early filtration as outlined before. Please see Figure 2 and Figure 3 for the results for each of the two analyzed documents.

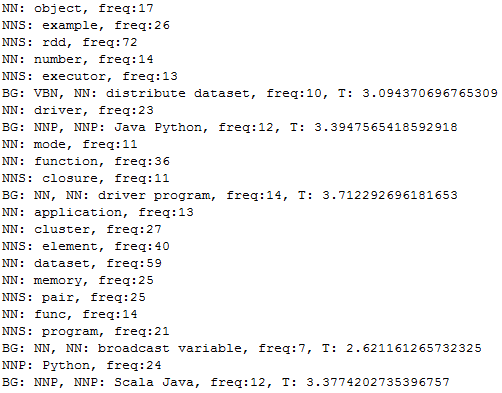


Figure 2: Noun extraction with early filtration results for the Spark Programming Guide

As one can see from the figures, these already contain quite a few useful concepts but useless tokens like “func” do appear somewhat frequently as well. This is where the scoring techniques come in. Also note that these extracted nouns are not scored so they are not yet ranked the appear in the order in which they appeared in the text.

## General purpose concept extraction

In my opinion the most useful form of concept extraction I developed is this general purpose concept extraction scheme. It returns very basic, simple unigrams mostly, and only a few general bigrams. This can be used to build a general understanding of the software without going into too much detail about the technicalities or the source code. The results of this method can be seen in Figure 4. Here, I used the top scorers from JATE2.0 using ATTF and C-value scores and then of those only considered the terms that occur within the set of early filtration nouns I got from my Java software. Why those two scores were chose should by apparent from my preceding discussion about the techniques. But they also return too general or source code related terms that do not serve as useful concepts for us like “Java”, “Spark” or “for(int i=0; i<length; i++)”. By considering only my early filtration terms, these too commonly occurring terms are filtered out and we get more useful concepts.

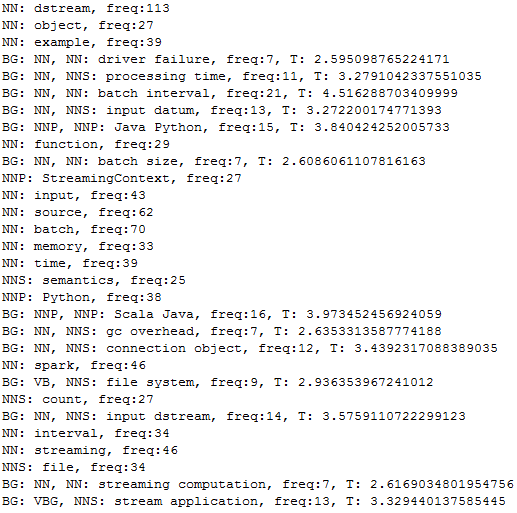


Figure 3: Noun extraction with early filtration results for the Spark Streaming Guide

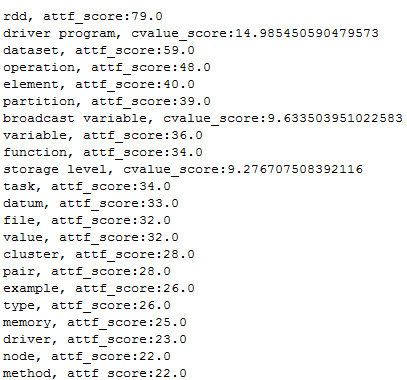


Figure 4: General purpose concept extraction scheme results for the Spark Programming Guide

## Source code related concepts extraction

In our endeavor to understand the source code underlying the analyzed documentation we will also often need more source code related concepts such as class names that we can use as pointers for the bug localization. The best scheme I could come up with for such source code related concepts uses GlossEx scoring. This works well to find “weird” terms such as “SparkContext” that are not to be found in the dictionary. The best scoring ones of these usually happen to be pretty good source code concepts. The results in Figure 5 clearly show that these source code concepts could mostly not be found in dictionaries. Since these terms are so “weird” they are usually not in the list of nouns from the dictionary I identified after my early filtration. Therefore, the only filtration I used on the terms here was removing those containing special characters since useful concepts usually do not contain any special characters.

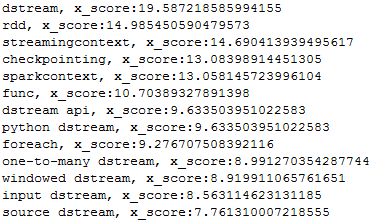


Figure 5: Source code related concepts extraction results for Spark Streaming Guide

## Technically-focused Concepts extraction

As mentioned before, RAKE also gave very good results for more technically-focused but not directly source code related concepts. Interestingly, the JATE2.0 implementation of RAKE seemed to give worse results than the original version in Python so I suggest using that version for extracting these types of concepts. The results of top scorers can be seem below.

('output operation', 3.909090909090909), ('external system', 3.8055555555555554), ('akka actor', 3.7142857142857144), ('advanced source', 3.7), ('batch size', 3.6904761904761907), ('streaming computation', 3.683168316831683), ('broadcast variable', 3.666666666666667), ('cluster resource', 3.65), ('sliding window', 3.5925925925925926), ('batch interval', 3.5870279146141213), ('stateful transformation', 3.571428571428571), ('input stream', 3.5625)

For similar reasons as before, the only filtration I did here was lemmatizing all words and removing duplicates. Otherwise, “output operations” (the plural) would be the second highest scoring term, for instance.

# Usage

My software is used by simply compiling the Eclipse project after setting the respective paths for the files to be analyzed and score results from JATE2.0. So, to make use of all functionality provided by my program one must supply the output of JATE2.0 in JSON-format which must be created manually. This can be done using the JATE library via a command like the following:

java -cp JATE\_JAR\_PATH uk.ac.shef.dcs.jate.app.AppCValue -corpusDir FOLDER\_WITH\_DOCUMENTS -o FILENAME.json SOLR\_TESTBED\_PATH ACLRDTEC

ACLRDTEC, here, is a specific configuration of Apache Solr and happens to give better results than the second standard configuration GENIA. A quickstart guide for using JATE2.0 can be found under <https://github.com/ziqizhang/jate/wiki/Quick-start>. Instead of AppCValue all the values from the table in Figure 6 can be used to choose the technique wanted. For Weirdness, GlossEx and TermEx, “-r SOLR\_TESTBED\_PATH/ACLRDTEC/conf/bnc\_unifrqs.normal” has to be added as well since this is the path were the dictionary of good words is located in your JATE2.0 distribution.

| **Algorithm** | **APP\_ALGORITHM** |
| --- | --- |
| TTF | uk.ac.shef.dcs.jate.app.AppTTF |
| ATTF | uk.ac.shef.dcs.jate.app.AppATTF. |
| TTF-IDF | uk.ac.shef.dcs.jate.app.AppTFIDF |
| RIDF | uk.ac.shef.dcs.jate.app.AppRIDF |
| CValue | uk.ac.shef.dcs.jate.app.AppCValue |
| ChiSquare | uk.ac.shef.dcs.jate.app.AppChiSquare |
| RAKE | uk.ac.shef.dcs.jate.app.AppRAKE |
| Weirdness | uk.ac.shef.dcs.jate.app.AppWeirdness |
| GlossEx | uk.ac.shef.dcs.jate.app.AppGlossEx |
| TermEx | uk.ac.shef.dcs.jate.app.AppTermEx |

Figure 6: JATE2.0 scoring algorithms

Currently, the main function in the source code corresponding to this document is configured to first analyze the files directly and doing the noun extraction with early filtration. After these results are printed, the program reads in the C-value and ATTF results from JATE and applies the filtration outlined above to output the aforementioned general purpose concepts. Finally, the JATE results of GlossEx are read in and minimally filtered before being printed out as the source code related concepts.

All of these steps can be modified use different filtration or read in different scoring files by simply changing the paths for the existing filtration schemes. Adaption of the source code should be pretty straightforward.

RAKE has to be used separately which is easiest by downloading the library with a sample program in Python from their website and then simply changing it to read in the files in question. This approach is completely independent of my Java program.

All of the extraction techniques, what POS patterns are considered, how the text is processed in regards to stop words and lemmatization/stemming, how different file types are indexed and many, many more things about JATE2.0 and Apache Solr can be can be configured in the schema.xml file located in the Solr testbed in the conf folder of the configuration used at the end of the cmd-command above. However, the standard settings are very good for most purposes.

If one wants to use no lemmatization and prefers OpenNLP, my OpenNLPEngine.java file can be made to replace the StanfordNLPEngine.java file by simply changing all the references in the main function. However, the OpenNLP version was not thoroughly tested and might cause errors in some cases.

## Usage summary

The Java program needs the Stanford CoreNLP 3.6 library and any version of the json.org java-json library to compile. If one wants to use the OpenNLP version, that library in version 1.6 will be required instead of the Stanford one. In addition, JATE2.0 has to be downloaded, configured and used as mentioned before to produce the scoring results. By changing the paths in the Java programs, those are read in and used to produce the above results when ran. My submission includes the JATE results for various settings and a JATE folder that is complete up to the src and solr-testbed folders since those are very big. The JATE jar is not included for the same reason either. The RAKE python project in my submission is also complete up to the “res” folder which contains the large dictionary and can be downloaded with the official Python RAKE implementation.